Chapter 6

Expressions and Operators

An expression is any series of symbols that VAX C uses to produce a value. The simplest expressions are constants and variable names, which contain no operators and yield a value directly. Other expressions combine operators and subexpressions to produce values.

In some instances, the compiler makes conversions so that the data types of the operands are compatible. This chapter refers to these rules as the arithmetic conversion rules. See Section 6.9.1 for more information about these rules.

This chapter discusses the following topics:

- The lvalues and rvalues
- Primary expressions and operators
- An overview of the VAX C operators
- Unary expressions and operators
- Binary expressions and operators
- The conditional expression and operator
- Assignment expressions and operators
- The comma expression and operator
- Data-type conversions

6.1 The lvalues and rvalues

A variable identifier is one of the primary VAX C expressions. (See Section 6.2 for more information about primary expressions.) This type of expression yields a single value, the contents of the variable. However, when using the variable identifier with other operators, the expression evaluates to the variable's location in memory. The address of the variable is the variable's lvalue. The object stored at that address is the variable's rvalue. For example, VAX C uses both the lvalue and the rvalue of variables in the evaluation of an expression as follows:

```
x = y;
```

The contents of variable y are taken and assigned to variable x. The expression on the right side evaluates to the variable's rvalue while the expression on the left side evaluates to the variable's lvalue when performing an assignment.

The following syntax defines the VAX C expressions that either have or produce lvalues:

```
lvalue ::=
      identifier
```

```
primary [ expression ]
lvalue . identifier
primary -> identifier
* expression
( lvalue )
```

These expressions represent the following identifiers and references:

- Identifiers of scalar variables, structures, and unions
- References to scalar array elements
- References to structure and union members, except for references to fields that are not lvalues
- References to pointers (also called dereferenced pointers; that is, an asterisk (*) followed by an address-valued expression)
- Any of the previous expressions enclosed in parentheses

All lvalue expressions represent a single location in a computer's memory. For a graphic explanation of lvalues and rvalues, see Chapter 4.

6.2 Primary Expressions and Operators

Simple expressions are called primary expressions, which denote values. Primary expressions include previously declared identifiers, constants (including strings), array references, function calls, and structure or union references. The syntax descriptions of the primary expressions are as follows:

```
primary ::=
        identifier
        constant
        string
        ( expression )
        primary ( expression-list )
        primary [ expression ]
        lvalue . identifier
        primary -> identifier
```

The simplest identifiers are variable names, and string or arithmetic constants. Other forms are expressions (delimited by parentheses), function calls, array references, lvalues and rvalues, and structure and union references.

6.2.1 Parenthetical Expressions

An expression within parentheses has the same type and value as the same expression without parentheses. As in declarations, you can delimit any expression using parentheses to change the grouping, or associative precedence, of the operators in a larger expression.

6.2.2 Function Calls

A function call is a primary expression followed by parentheses. The parentheses can contain a list of arguments (separated by commas) or it can be empty. An undeclared function is assumed to be a function returning **int**. If you declare an identifier as a function returning **int**, but use the identifier in a context other than a function call, it converts to the address of a function returning.

Consider the following declaration:

```
double atof();
```

The previous example declares a function returning **double**. You can then use the identifier **atof** in a function call, as in the following example:

```
result = atof(c);
```

You can also use the **atof** identifier in other contexts without the parentheses, as follows:

```
dispatch(atof);
```

The **atof** identifier converts to the address of that function, and the address is passed to the function dispatch.

Functions can also be called using a pointer to a function. Consider the following pointer declaration and assignment:

```
double (*pfd) ( );
    .
    .
    .
pfd = atof;
```

To call the function, you can specify the following form:

```
result = (*pfd) (c);
```

VAX C also accepts a pointer to a function, as shown in the following form:

```
result = pfd (c);
```

While the first call to the function is valid, the second call to the function is simpler and requires fewer keystrokes.

6.2.3 Array References ([ ])

Use bracket operators ([ ]) to see elements of arrays. In an array defined as having three dimensions, you refer to a specific element within the array, as in the following example:

```
int sample_array[10][5][2];        /* Array declaration       */
int i = 10;
sample_array[9][4][1] = i;         /* Assign value to element  */
```

This example assigns a value of 10 to element sample_array[9][4][1].

In addition, if an array reference is not fully qualified, it refers to the address of the first element in the dimension that is not specified. Consider the following statement:

```
sample_array[9][4] = 10;
```

This statement assigns a value of 10 to the element sample_array[9][4][0].

Consider the following statement:

```
sample_array = 10;
```

The statement assigns a value of 10 to the element sample_array[0][0][0]. A reference to an array name with no bracket operator is often used to pass the array's address to a function, as in the following case:

```
funct(array);
```

You can also use bracket operators to perform general address arithmetic, using the following form:

```
addr[intexp]
```

In the previous example, addr is the address of some previously declared object (pointer-valued) and the variable intexp is an integer-valued expression. The result of the expression is scaled, or multiplied, by the size, in bytes, of the addressed object. If intexp is a positive integer, the result is the address of a subsequent object of this size. If intexp is 0, the result is the address of the same object. If intexp is negative, the result is the address of a previous object. The expressions *(addr + intexp) and addr[intexp] are equivalent because both expressions reference the same memory location.

6.2.4 Structure and Union References

You can reference a member of a structure or union with either the period (.) or the right arrow (–>) operator.

A primary expression followed by a period and an identifier refers to a member of a structure or union, and is itself a primary expression. The first expression must be an lvalue naming a structure or union. The identifier must name a member of that structure or union. The result is a reference (if the member is a scalar) to the named member of the structure or union. The name of the desired member must be preceded by a period-separated list of the names of all higher level members. For more information about structures and unions, see Chapter 7.

The form for a pointer to a structure and union uses the right-arrow operator (–>), which is specified with a hyphen (–) and a greater-than symbol (>). A primary expression followed by a right arrow and an identifier refers to a member of a structure or union. The first expression must be a pointer to a structure or a union. The identifier following the right-arrow operator must name a declared member of that structure or union. The result is a reference to the named member.

The primary expression in bboth cases can be either a pointer or an integer. If it is a pointer, VAX C assumes that it points to a structure where the name on the right is a member. If it is an integer, VAX C assumes that it is the absolute address of the appropriate structure in machine storage units. If you specify something other than a pointer to a structure or union, VAX C signals the QUALNOTSTRUCT informational message. If you point to a different structure or union type, VAX C signals the NONSEQUITUR informational message.

6.3 Overview of the VAX C Operators

You can use the simpler variable identifiers and constants in conjunction with VAX C operators to create more complex expressions. Table 6–1 lists the VAX C operators.

Table 6–1: VAX C Operators

Operator	Example	Result
- [unary]	-a	Negative of a
* [unary]	*a	Reference to object at address a
& [unary]	&a	Address of a
~	~a	One's complement of a
++ [prefix]	++a	a after increment
++ [postfix]	a++	a before increment
- - [prefix]	- -a	a after decrement
- - [postfix]	a- -	a before decrement
sizeof	sizeof(t1)	Size, in bytes, of type t1
	sizeof e	Size, in bytes, of expression e
(type-name)	(t1)e	Expression e, converted to type t1
+	a + b	a plus b
- [binary]	a - b	a minus b
* [binary]	a * b	a times b
/	a / b	a divided by b
%	a % b	remainder of a/b (a modulo b)
>>	a>> b	a, right-shifted b bits
<<	a << b	a, left-shifted b bits
<	a < b	1 if a < b; 0 otherwise
>	a > b	1 if a > b; 0 otherwise
<=	a <= b	1 if a <= b; 0 otherwise
>=	a >= b	1 if a >= b; 0 otherwise
==	a == b	1 if a equal to b; 0 otherwise
!=	a != b	1 if a not equal to b; 0 otherwise
& [binary]	a & b	Bitwise AND of a and b
|	a | b	Bitwise OR of a and b
^	a ^ b	Bitwise XOR (exclusive OR) of a and b
&&	a && b	Logical AND of a and b (yields 0 or 1)
| |	a | | b	Logical OR of a and b (yields 0 or 1)
!	!a	Logical NOT of a (yields 0 or 1)
?:	a ? e1 : e2	Expression e1 if a is nonzero, Expression e2 if a is zero
=	a = b	a (with b assigned to a)
+=	a += b	a plus b (assigned to a)
-=	a -= b	a minus b (assigned to a)
*=	a *= b	a times b (assigned to a)
/=	a /= b	a divided by b (assigned to a)
%=	a %= b	Remainder of a/b (assigned to a)
>>=	a >>= b	a, right-shifted b bits (assigned to a)
<<=	a <<= b	a, left-shifted b bits (assigned to a)
&=	a &= b	a AND b (assigned to a)
| =	a | = b	a OR b (assigned to a)
^=	a ^= b	a XOR b (assigned to a)
,	e1,e2	e2 (e1 evaluated first)

The operators fall into the following categories:

- Unary operators, which take a single operand
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression

- Assignment operators, which assign a value to a variable, optionally performing an additional operation before the assignment takes place
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions
- Primary operators, which usually modify or qualify identifiers (see Section 6.2 for more information)

Table 6–2 presents the precedence by which the compiler evaluates operations. Operators with the highest precedence appear at the top of the table; operators with the lowest precedence appear at the bottom. Operators of equal precedence appear in the same row.

Table 6–2: Precedence of VAX C Operators

Category	Operator	Associativity
Primary	( ) [ ] -> .	Left to right
Unary	! ~ ++ - - (type) * & **sizeof**	Right to left
Binary (multiplication)	* / %	Left to right
Binary (addition)	+ –	Left to right
Binary (shift)	<< >>	Left to right
Binary (relational)	< <= > >=	Left to right
Binary (equality)	== !=	Left to right
binary (bitand)	&	Left to right
Binary (bitxor)	^	Left to right
Binary (bitor)	|	Left to right
Binary (AND)	&&	Left to right
Binary (OR)	||	Left to right
Conditional	?:	Right to left
Assignment	= += –= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Consider the following expression:

```
A*B+C
```

The identifiers A and B are multiplied first because the multiplication operator (*) is of higher precedence than the addition operator (+). The associative rule applies to each row of operators. The following expression evaluates as A divided by B with the result then divided by C, because the division operator evaluates from left to right:

```
A/B/C
```

6.4 Unary Expressions and Operators

You form unary expressions by combining a unary operator with a single operand. All unary operators are of equal precedence and group from right to left. They perform the following operations:

- Negate a variable arithmetically (–) or logically (!)
- Increment (++) and decrement (– –) variables
- Find addresses (&) and dereference pointers (*)
- Calculate a one's complement (~)
- Force the conversion of data from one type to another (the cast operator)
- Calculate the sizes of specific variables or types (**sizeof**)

6.4.1 Negating Arithmetic and Logical Expressions

Consider the syntax of the following expression:

```
- expression
```

This defines the arithmetic negative of expression. The compiler performs the arithmetic conversions. The negative of an **unsigned** quantity is computed by subtracting its value from 2^{32}. There is no unary plus operator in VAX C.

The result of the following expression is the logical (Boolean) negative of the expression:

```
!  expression
```

If the result of the expression is 0, the negated result is 1. If the result of the expression is not 0, the negated result is 0. The type of the result is **int**. The expression can be a pointer (or other address-valued expression) or an expression of any arithmetic type.

6.4.2 Incrementing and Decrementing Variables

The object that the lvalue refers to in the following expression is incremented before its value is used:

```
++lvalue
```

After evaluating this expression, the result is the incremented rvalue, not the corresponding lvalue. For this reason, expressions that use the increment and decrement operators in this manner cannot appear by themselves on the left side of an assignment expression that needs an lvalue.

The object to which the lvalue refers in the following expression increments after its value is used:

```
lvalue++
```

The expression evaluates to the value of the object before the increment, not the incremented variable's lvalue.

If the operand is a pointer, the address is incremented by the length of the addressed object, not by the integer value 1.

The objects of the following lvalues point to other variables:

```
--lvalue
lvalue--
```

These pointers decrement not by the integer value 1, but by the length of the addressed object. The data type of the variable determines the amount of the increment or decrement. If declared as a pointer, the variable increments or decrements by the length determined by the addressed object's data type. For example, incrementing a pointer to **int** increments the value of the pointer by 4. If declared as an integer, the variable increments or decrements by the value 1.

When using the increment and decrement operators, do not depend upon the order of evaluation of expressions. Consider the following ambiguous expression:

```
k = x[j] + j++;
```

Is the value of variable j in x[j] evaluated before or after the increment occurs? Do not make assumptions about which expressions the compiler evaluates first. To avoid ambiguity, increment the variable in a separate statement.

6.4.3 Computing Addresses and Dereferencing Pointers

Consider the following syntax:

& variable

The expression results in the lvalue (address) of variable. You may not apply the ampersand operator (&) to **register** variables or to bit fields in structures or unions.

NOTE

> In VAX C, the compiler changes any **register** variable to which the ampersand operator applies to **auto**. The compiler issues no warning message unless you use **–V standard=portable**; if you do, the compiler issues an appropriate message.

In the special context of argument lists, you may apply the ampersand operator to constants. This use of the ampersand operator passes constants to user-defined functions that expect arguments to be passed by reference. This use is not recommended for other applications. It is VAX C-specific and not portable.

Since function identifiers and unqualified array identifiers are lvalues, you cannot apply the ampersand operator to these identifiers. If you apply the ampersand operator to function identifiers or to unqualified array identifiers, VAX C considers this a redundant use of the ampersand operator and generates the appropriate error message when the **–V**"STANDARD=NOPORTABLE" option is used.

When an expression evaluates to an address, as in the following example, the address is used to indirectly access the object that the address refers to:

```
* pointer
```

An expression using the indirection operator (*) evaluates to the object pointed to by a pointer or by an address-valued expression.

6.4.4 Calculating a One's Complement

Consider the following syntax:

```
~ expression
```

The result is the one's complement of the evaluated expression; it converts each 1-bit into a 0-bit and each 0-bit into a 1-bit. The expression must be integral (an integer or character). The compiler performs necessary arithmetic conversions.

6.4.5 Forcing Conversions to a Specific Type Using the Cast Operator

The cast operator forces the conversion of an operand to a specified scalar data type. The operator consists of a data-type name, in parentheses, which precedes the operand expression, as follows:

```
(type-name) expression
```

The resulting value of the expression converts to the named data type, just as if the expression were assigned to a variable of that type. If the operand is a variable, its value converts to the named type. The variable's contents do not change. The type name has the following syntax:

```
type-name ::=
      type-specifier abstract-declarator
```

In simple cases, type-specifier is the keyword for a data type, such as **char** or **double**. The identifier type-specifier may also be a structure, union specifier, an **enum** specifier, or a **typedef** tag.

An abstract-declarator in a parameter declaration is a declaration without an identifier or data-type keyword as follows:

```
abstract-declarator ::=
      empty
      ( abstract-declarator )
      * abstract-declarator
      abstract-declarator ( )
      abstract-declarator [ constant-expression ]
```

Consider the following form of the abstract-declarator:

```
abstract-declarator( )
```

To avoid confusion with the previous form, the abstract-declarator may not be empty in the following form:

```
(abstract-declarator)
```

Abstract declarators can include the brackets and parentheses that indicate arrays and function calls. However, cast operations cannot force the conversion of any expression to an array, function, structure, or union. The brackets and parentheses are used in such operations as the following, which casts identifier P1 to pointer to array of **int**:

```
(int (*)[]) P1
```

This kind of cast operation does not change the contents of P1; it only causes the compiler to treat the value of P1 as a pointer to such an array. For example, casting pointers in this manner can change the scaling that occurs when you add an integer to a pointer.

6.4.6 Calculating Sizes of Variables and Data Types (sizeof)

Consider the following syntax:

sizeof expression
sizeof (type-name)

The result is the size, in bytes, of the operand. In the first case, the result of **sizeof** is the size determined by the declarations of the objects in the expression. In the second case, the result is the size, in bytes, of an object of the named type. The syntax of type-name is the same as that for the cast operator. See Section 6.4.5 for more information about the cast operator.

6.5 Binary Expressions and Operators

The binary operators are categorized as follows:

- Additive operators: addition (+) and subtraction (–)
- Multiplication operators: multiplication (*), mod (%), and division (/)
- Equality operators: equality (+ =) and inequality (!=)
- Relational operators: less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=)
- Bitwise operators: AND (&), OR (|), and XOR (^)
- Logical operators: AND (&&) and OR (| |)
- Shift operators: left shift (<<) and right shift (>>)

The following sections describe these binary expressions and operators.

6.5.1 Additive Operators

The additive operators (+) and (–) perform addition and subtraction. Their operands are converted, if necessary, following the arithmetic conversion rules. For more information, see Section 6.9.1.

You can increment an array pointer by adding an integral variable to the address of an array element. The compiler calculates the size of one array element, multiplies that by the integer to obtain the offset value, and then adds the offset value to the address of the designated element. For example:

```
int arr[10];
int *p = arr;
p = p + 1; /* Increments by 4 */
```

You may subtract a value of any integral type from a pointer or address; in that case, the same conversions apply as for addition.

When you add or subtract two **enum** constants or variables, the type of the result is **int**.

If you subtract two addresses of objects of the same type, the result is an **int** representing the number of objects separating the addressed objects. The result of this conversion is unpredictable unless the two objects are in the same array.

6.5.2 Multiplication Operators

The multiplication operators (*), (/), and (%) perform arithmetic conversions, if necessary. The binary operator (*) performs multiplication. The binary operator (/) performs division. When integers are divided, truncation is toward 0.

The binary mod operator (%) divides the first operand by the second and yields the remainder. Both operands must be integral. The sign of the result is the same as the sign of the quotient. If variable b is not 0, then the following is always true:

```
(a/b)*b + a%b = a
```

6.5.3 Equality Operators

The equality operators equal-to (==) and not-equal-to (!=) perform the necessary arithmetic conversions on their two operands. These operators produce a result of type **int**. Consider the following example:

```
a<b == c<d
```

The result is the value 1, if both relational expressions have the same truth value, and 0 if they do not.

Two pointers or addresses are equal if they identify the same storage location. You can compare a pointer or address with an integer, but the result is not portable unless the integer is 0. A null pointer is considered equal to 0.

Although different symbols are used for assignment and equality, (=) and (==), respectively, VAX C allows either operator in some contexts, so you must be careful not to confuse them. Consider the following example:

```
if (x=1) statement-1;
else     statement-2;
```

In the previous example, statement-1 always executes, since the result of assignment x=1 delimited by parentheses is equivalent to the value of x, which is equal to 1 or true.

6.5.4 Relational Operators

The relational operators compare two operands and produce a result of type **int**. The result is the value 0 if the relation is false, and 1 if it is true. The operators are less-than (<), greater-than (>), less-than or equal-to (<=), and greater-than or equal-to (>=). The compiler performs necessary arithmetic conversions.

If you compare two pointers or addresses, the result depends on the relative locations of the two addressed objects. Pointers to objects at lower addresses are less than pointers to objects at higher addresses. If two addresses indicate elements in the same array, the address of an element with a lower subscript is less than the address of an element with a higher subscript.

The relational operators group from left to right. However, note that the following statement compares the variable c with 0 or 1 (the possible results of a<b); it does not mean "if b is between a and c...":

```
if (a<b<c)...
```

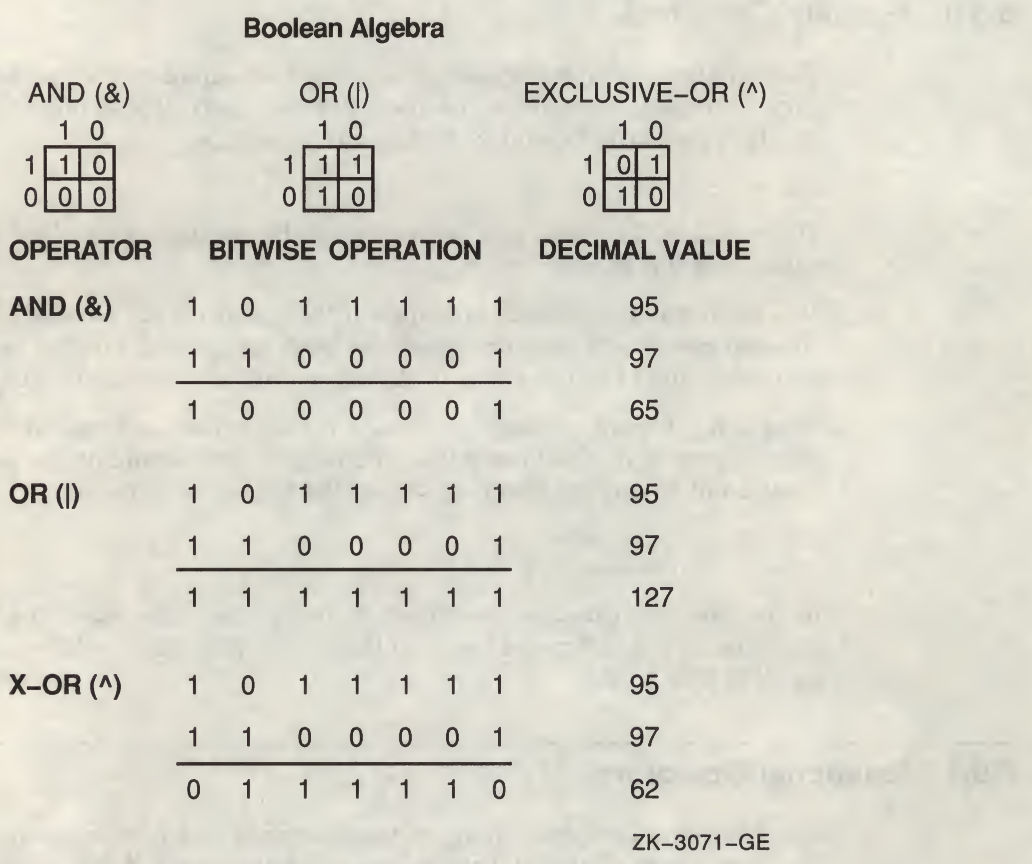
In order to check that b is between a and c, you should use the following code:

```
if (a<b && <c)...
```

6.5.5 Bitwise Operators

You may only use bitwise operators with integral operands, with variables of types **char** and **int** (all sizes). The compiler performs the necessary arithmetic conversions. The result of the expression is the bitwise AND (&), OR (|), or EXCLUSIVE OR, which is represented by XOR (^), of the two operands. The compiler always evaluates all operands. Figure 6–1 shows the effects of Boolean algebra when using the bitwise operators.

Figure 6–1: Boolean Algebra and the Bitwise Operators



Boolean Algebra

AND (&)

	1	0
1	1	0
0	0	0

OR (|)

	1	0
1	1	1
0	1	0

EXCLUSIVE–OR (^)

	1	0
1	0	1
0	1	0

OPERATOR	BITWISE OPERATION	DECIMAL VALUE
AND (&)	1 0 1 1 1 1 1	95
	1 1 0 0 0 0 1	97
	1 0 0 0 0 0 1	65
OR (|)	1 0 1 1 1 1 1	95
	1 1 0 0 0 0 1	97
	1 1 1 1 1 1 1	127
X–OR (^)	1 0 1 1 1 1 1	95
	1 1 0 0 0 0 1	97
	0 1 1 1 1 1 0	62

ZK–3071–GE

In Boolean algebra, VAX C compares values bit by bit. If you use the bitwise AND, and you compare a bit value 1 and a bit value 0, the result is 0. When you use the bitwise AND, both compared bits must be 1, as shown in Figure 6–1, for the result to be 1. When you use the bitwise OR, either bit value can be 1 for the result to be 1. When you use the bitwise EXCLUSIVE OR, either value, but not both, can be 1 for the result to be 1. Figure 6–1 shows the use of all three bitwise operators on two common values.

6.5.6 Logical Operators

The logical operators are AND (&&) and OR (| |). These operators guarantee left-to-right evaluation. The result of the expression (of type **int**) is either 0 (false) or 1 (true). If the compiler can determine the result by examining only the left operand, it does not evaluate the right operand. Consider the following expression:

```
E1 && E2
```

The result is 1 if both its operands are nonzero, or 0 if one operand is 0. If expression E1 is 0, E2 is not evaluated. Similarly, in the following expression, the result is 1 if either operand is nonzero, and 0 otherwise:

```
E1 || E2
```

If expression E1 is nonzero, E2 is not evaluated.

The operands of logical operators need not have the same type, but each must be one of the fundamental types or must be a pointer or other address-valued expression.

6.5.7 Shift Operators

The shift operators (<<) and (>>) take two operands, which must be integral. The compiler performs the necessary arithmetic conversions on both operands. The right operand is then converted to **int**, and the type of the result is the type of the left operand. Consider the following example:

```
E1 << E2
```

The result is the value of expression E1 shifted to the left by E2 bits. The compiler clears vacated bits.

The result of the following expression is the value of expression E1 shifted to the right by E2 bits:

```
E1 >> E2
```

The compiler clears vacated bits if E1 is **unsigned**; otherwise, bits are filled with a copy of E1's sign bit.

The result of the shift operation is undefined if the right operand (E2 in the previous example) is negative or is greater than 32.

6.6 The Conditional Expression and Operator

The conditional operator (?:) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. Consider the following example:

```
E1  ?  E2  :  E3
```

If expression E1 is nonzero (true), then E2 is evaluated. If E1 is 0 (false), E3 is evaluated. Conditional expressions group from right to left. The compiler makes conversions in the following order:

- If possible, the arithmetic conversions are performed on expressions E2 and E3, so that they will result in the same type.
- Otherwise, if expressions E2 and E3 are address expressions indicating objects of the same type, the result has that type.

- Otherwise, either one of the E2 and E3 operands must be an address expression, and the other, the constant 0. The result has the type of the addressed object.

6.7 Assignment Expressions and Operators

An assignment is an expression as well as an operation. The result of an assignment expression is the value of the target variable after the assignment. You can use assignments as subexpressions in larger expressions.

The set of assignment operators consists of the equal sign (=) alone and in combination with binary operators. An assignment expression has two operands (an lvalue and an expression separated by one of these operators). The following two assignment expressions are identical:

```
E1 += E2;
```

```
E1 = E1 + E2;
```

The expression E1 is evaluated once and must result in an lvalue. The type of the assignment expression is the type of E1, and the result is the value of E1 after the operation is finished. You must delimit some expressions in parentheses if the expressions possibly contain other operators of a lower precedence. For example, the following expressions produce identical results:

```
a *= b + 1;
```

```
a = a * (b + 1);
```

However, the following example produces different results:

```
a = (a * b) + 1;
```

In the following simple assignment expression, the value of expression E2 replaces the previous object of E1:

```
E1 = E2
```

In the following example, the expression adds 100 to the contents of a_number[1]:

```
a_number[1] += 100;
```

The result of the expression is the result of the addition, which has the same type as a_number[1].

If both assignment operands are arithmetic, the right operand is converted to the type of the left operand before the assignment. (See Section 6.9.1.)

You can use the assignment operator (=) to assign values to structure and union members. You can assign one structure value to another if you define the structures to be the same size. With all other assignment operators, all right operands and all left operands must be either pointers or evaluate to arithmetic values. If the operator is (-=) or (+=), the left operand can be a pointer, and the right operand (which must be integral) is converted in the same manner as the right operand in the binary plus (+) and minus (-) operations.

You can assign an address to an integer, an integer to a pointer, and the address of an object of one type to a pointer of another type. These assignments are simple copy operations, with no conversions. This usage can cause addressing exceptions when you use the resulting pointers. However, if the constant *0 is assigned* to a pointer, the result is a null pointer. The null pointer is distinguishable (by the equality operators) from a pointer that points to any object.

For the sake of compatibility with other C implementations, VAX C allows certain exceptions to the spellings of the compound assignment operators shown in Table 6–2. The exceptions are as follows:

- If you write the operators in the order shown in Table 6–2, you can separate the two characters with blank spaces. The following two examples produce the same results:

  ```
  E1 += E2;
  ```

  ```
  E1 + = E2;
  ```

- You can also write the operators with the characters in reverse order, as in the following example:

  ```
  E1 =+ E2;
  ```

The second form generates an informational message for the following reasons:

- The syntax allowed by VAX C is more restrictive in this case. Specifically, the characters (*, –, and &) must be immediately adjacent to the equal sign (=) character because they also appear in unary operators. This placement avoids ambiguities in cases such as the following, which multiplies the result of expression E1 by the value of p:

  ```
  E1 =*p;
  ```

- Even with usage that follows the guidelines, you can introduce ambiguities, as in the following example:

  ```
  E1 =/*part of a comment...
  ```

6.8 The Comma Expression and Operator

If you separate two expressions with the comma operator, they evaluate from left to right, and the compiler discards the result of the left expression. If you separate many expressions with commas, the compiler discards all but the result of the rightmost expression. The following example assigns the value 1 to variable R and the value 2 to variable T:

```
R = T = 1,   T += 2,   T -= 1;
```

The type and value of the result of a comma expression are the type and value of the right operand. The operator evaluates from left to right.

You must delimit comma expressions with parentheses if they appear where commas have some other meaning, as in argument and initializing lists. Consider the following example:

```
f(a, (t=3,t+2), c)
```

The function f is called with the arguments a, 5, and c. In addition, variable t is assigned the value 3.

6.9 Data-Type Conversions

VAX C performs data-type conversions in the following situations:

- When two or more operands of different types appear in an expression (including an assignment)
- When arguments other than long integers, addresses, or double-precision, floating-point numbers are passed to a function

- When arguments that do not conform exactly to the parameters declared in a function prototype are passed to a function
- When the data type of an operand is deliberately converted by the cast operator (See Section 6.4.5 for more information on the cast operator)

6.9.1 Converting Operands

The following rules (referred to as the arithmetic conversion rules) govern the conversion of operands in arithmetic expressions. Although they do not specify explicit conversions at the machine-language level, the rules govern in the following order:

- Any operands of type **char** or **short** (signed or unsigned) convert to their 32-bit equivalents (**int** or **unsigned int**), and any of type **float** convert to **double**.
- If either operand is **double**, the other converts to **double**, and that is the type of the result, unless you specify the **-f** option with the **vcc** command.
- If either operand is **unsigned**, the other converts to **unsigned**, and that is the type of the result.
- Otherwise, both operands must be **int**, and that is the type of the result.

The arithmetic conversions are performed on all arithmetic operands. Some operators, such as the shift operators (>> and <<) require integers as operands. If one operand is of type **float** or **double**, you cannot meet this requirement.

In previous versions of VAX C under the VMS operating system, floating-point arithmetic was carried out in double precision. Since the proposed ANSI C standard no longer requires this conversion, VAX C performs arithmetic in single precision if you specify the **-f** option with the **vcc** command. Whenever an operand of type **float** appears in an expression and **-f** is specified, it is treated as a single-precision object — unless the expression also involves an object of type **double**, in which case the usual arithmetic conversion applies.

When you convert an operand of type **double** to **float**, (for example, by an assignment) the compiler rounds the operand before truncating it to **float**.

The compiler may convert a **float** or **double** value operand to an integer by assignment to an integral variable. In VAX C, the truncation of the **float** or **double** value is always toward 0.

Conversions also take place between the various kinds of integers. In VAX C, variables of type **char** are bytes treated as signed integers. When a longer integer is converted to a shorter integer or to **char**, it is truncated on the left; excess bits are discarded. Consider the following example:

```
int i;
char c;

i = 0xFFFFFF41;
c = i;
```

The result is to assign hex 41 ('A') to variable c. The compiler converts shorter signed integers to longer ones by sign extension.

When the compiler combines an unsigned integer and a signed integer, the signed integer converts to **unsigned** and the result is **unsigned**. All conversions from signed to unsigned perform an intermediate conversion to **int**. For example, the compiler converts a **char** or **short** operand to an unsigned version by first converting it to a signed **int** and then by truncating it to form the unsigned version. All conversions from unsigned to signed (such as by the cast operator) involve an intermediate conversion to **unsigned int**.

You can also add integers to pointers, in which case the integer is scaled (multiplied) by a factor that depends on the type of the object that the pointer points to. See Section 6.5.1 for more information about scaling pointers.

6.9.2 Converting Function Arguments

The data types of function arguments are assumed to match the types of the formal parameters unless a function prototype declaration is present. In the presence of a function prototype, all arguments in the function invocation are compared for assignment compatibility to all parameters declared in the function prototype declaration. If the type of the argument does not match the type of the parameter but is assignment compatible, VAX C converts the argument to the type of the parameter. (See Section 6.9.1.) If an argument in the function invocation is not assignment compatible to a parameter declared in the function prototype declaration, VAX C generates an error message.

If there is no function prototype for the function, all arguments of type **float** convert to **double**, all variables of type **char** and **short** convert to **int**, all variables of type **unsigned char** and **unsigned short** convert to **unsigned int**, and an array or function name converts to the address of the named array or function. The compiler performs no other conversions automatically, and any mismatches after these conversions are programming errors.

Use the cast operator to pass arguments to parameters of different types. See Section 6.4.5 for more information on the cast operator. For more information about manipulating argument lists, see Chapter 4.